Packages, Extensions and EXEC in Comal80

Written for the Comal80 Development Group
by Arne Christensen, Metanic ApS, March 1983

This note comments on the inclusion or exclusion of the keyword EXEC in Comal80, on packages, and on extensions. The last one mentioned is a technical term for a new feature which Metanic has included in the latest revision of Metanic COMAL-80.

Inclusion or Exclusion of EXEC

The subject of our discussion in the Comal80 Standardisation Group was whether the EXEC keyword should be included or excluded from the Kernel definition. As Mogens Pelle and I pointed out, the question is a bit more complicated because we also have to consider the parantheses around the actual parameters in the procedure call statement. In addition we should consider syntax similarities with other Comal statements. This in order to ensure a consistent syntax.

I think that unstructured and structured statements differ considerably in syntax. This comes from the fact that only structured statements contain other statements which also has brought along the use of binding keywords like THEN. Besides, only structured statements are divided into multiple lines.

This means that I only consider comparison of procedure call statements with unstructured statements relevant. An exception is comparison with the procedure declaration statement.

In the following I will give examples of all the unstructured statements defined in Comal80 so far, except the ZONE statement which has not been agreed upon yet:

1.
```
STOP
CLOSE
RESTORE
GOTO FAILURE
RETURN BOXES
DELETE "EGG_FILE"
DATA 234, 629, "STRING"
DIM ADDRESS$ OF 50, EGG_ARRAY(0:100)
```

2.
```
INPUT "TYPE NOW: ": INITIAL_EGGS, EGGS_PER_BOX
READ EGGS_PER_BOX, MAX_BOXES
READ FILE 5, CURRENT_EGG: EGG_SUPPLIER$, ADDRESS$
WRITE FILE 5: EGG_ARRAY
PRINT EGG_SUPPLIERS, " HAS NOT"; " DELIVERED YET"
PRINT USING "#####  ####  ": BOXES, SMASHED,
```

3.
```
SELECT OUTPUT "LP:"
OPEN FILE 5, "EGG_FILE", RANDOM EGG_RECORD_SIZE
CLOSE FILE 5
```

4.
```
FAILURE:
EGGS:=BOXES*EGGS_PER_BOX; SMASHED:=INITIAL_EGGS-EGGS
```

In the unstructured statements a rather general pattern of expression shows up. The simplest examples of this pattern are found in group 1: A keyword (which identifies the type of statement) followed by a list of items operated upon by the statement. The items are separated by commas.

In group 2 an extension of this pattern is found. Here, the list of items may be preceded by some auxiliary information followed by a colon. The syntax of the auxiliary information is varying. In addition, the PRINT and PRINT USING statements employ semicolon as a separator besides the comma, and both may end in a separator.

In groups 3 and 4 I have collected some deviations from the pattern. There are some "double keywords" and deviations in the use of separators, most notably the assignment symbol ":=". More important, there is no leading keyword in the statements in group 4.

The important thing here is not the deviations from the pattern described, but the pattern itself.

Below, the possibilities for the syntax of a procedure call statement are outlined and compared with the general pattern:

```
(a)     EXEC CALCULATE_BOXES(EGGS, EGGS_PER_BOX, BOXES)
(b)     EXEC CALCULATE_BOXES EGGS, EGGS_PER_BOX, BOXES
(c)     CALCULATE_BOXES(EGGS, EGGS_PER_BOX, BOXES)
(d)     CALCULATE_BOXES EGGS, EGGS_PER_BOX, BOXES
```

(b) violates the pattern since there is no comma between CALCULATE_BOXES and EGGS. I do not think that it is a good idea to insert a comma. The procedure name must stand out as something special; otherwise it will result in confusion.

In (a) a delimiter has been inserted, although it is not a comma as usual. The closing paranthesis is necessary to balance the statement. The justification for using an opening paranthesis as delimiter (which is a unique approach in Comal) is that the procedure call statement becomes more like the procedure declaration statement. This can make it easier for beginners to understand the technique of formal and actual parameters.

(d) follows the general pattern closely. The procedure name acts as a keyword which is followed by the actual parameters separated by commas.

(c) is the proposal which started the discussion. It does not conform to the pattern, but is so close to it that it may cause considerable confusion. This is described below.

Personally, I like (a) and (d) best. I like (a) because of the similarity with the procedure declaration statement and because the EXEC keyword makes it completely clear that a description of the statement cannot be found in any manual or tutorial, but instead must be found in the program text itself. Also, a procedure call is syntactically similar to a function call.

I like (d) because it conforms to the general pattern of Comal statements, and because it makes it easy to add new types of statements (new "keywords") by means of library procedures and/or packages. This can be done without confusion on the part of the (novice) user, because the new statements conform to the general rules on how to form Comal statements. In fact, a novice can use a Comal that has been "augmented" in this way without distinguishing between original Comal statements and local augmentations.

If approach (c) is adopted, confusion will soon arise in applications where Comal is "augmented", since the paranthesis is required in some statements, but forbidden in others.

"Extensions"

In Metanic COMAL-80 approach (a) above has been adopted but approach (d) is used for "extensions". "Extensions" are routines written in assembler which define procedures, functions and operators. These can be used in a COMAL-80 program in quite the same way as standard statements, functions, and operators are used.

"Extensions" must be loaded (by means of a special command) before COMAL-80 statements employing them can be typed in. This makes possible a more effective check on the number of parameters and the parameter types already at the time of program entry and editing. Since COMAL-80 aims at the educational market, this is considered very important.

"Extensions" have some advantages over packages as proposed by TCD. Besides the already mentioned stronger parameter checking, it is possible to allow for a wider variety of parameter types, since the parameter passing mechanism can be different from the one used for procedures and functions written in Comal. In COMAL-80 "extensions" can have formal parameters of type "anytype", which means that the actual parameter can have any type. Information on the type of the actual parameter is automatically transferred to the "extension" in this case. Likewise, a formal parameter can have "any dimension", which is useful for implementing statements and functions which operate on any simple variable or array. Also, other conveniencies have been implemented.

The TCD Package Proposal

I would like to call attention to one of the reasons for the introduction of packages mentioned in the TCD package proposal. In the proposal it is stated that packages are "a means of providing a sensible overlay facility". I think there is a flaw in this argument. Overlay facilities are likely only to be attractive on the current 8-bit microprocessors, since on larger machines store will be abound. Of course, programs will always be written which will overflow any machine, but I do not think that programs that large will be written in Comal because of Comal's lack of data structuring facilities (records, user-defined types and pointers).

On 8-bit microprocessors, implementations of packages are likely to consume so much of the moderate store that only very little is left to the user. In this case it is no use to be able to part a big program since even the parts will be too big for the available store.

I conclude that packages will not be used as an overlay facility. The other reasons for using packages are, hovever, realistic, and justify the introduction of packages.

If packages are not used as overlays, then the ability to load and discard packages dynamically is not needed. I propose instead that the packages be loaded before statements which use the package facilities are typed in. The reasons for this I have described above in the discussion of "extensions".